Model-based Specification and Validation of Security and Dependability Patterns

Brahim Hamid, Christian Percebois

Model-Based Specification and Validation of Security and Dependability Patterns

Brahim Hamid(✉) and Christian Percebois

IRIT, University of Toulouse, 118 Route de Narbonne,
31062 Toulouse Cedex 9, France
{hamid,percebois}@irit.fr

Abstract. The requirement for higher Security and Dependability (S&D) of systems is continuously increasing, even in domains traditionally not deeply involved in such issues. In our work, we propose a modeling environment for pattern-based secure and dependable embedded system development by design. Here we study a general scheme for representing security and dependability (S&D) design patterns whose intention specification can be defined using a set of local properties. We propose an approach that associates Model Driven Engineering (MDE) and formal validation to get a common representation to specify patterns for several domains. The contribution of this work is twofold. On the one hand, we use model-based techniques to capture a set of artifacts to specify patterns. On the other hand, we introduce a set of artifacts for the formal validation of these patterns in order to guarantee their correctness. As an illustration of the approach, we study the authorization pattern.

Keywords: Pattern · Meta-model · Domain · Formalization · Model driven engineering · Security and Dependability

1 Introduction

Extra-functional concerns have become a strong requirement on one hand, and on the other hand more and more difficult to achieve, even in safety critical systems. They can be found in many application sectors such as automotive, aerospace and home control. Such systems come with a large number of common characteristics, including real-time, temperature, computational processing and power constraints and/or limited energy and common extra-functional characteristics such as dependability and security [1] as well as efficiency.

The integration of such concerns, for instance security, safety and dependability, requires the availability of both application development and concerns expertise at the same time. Therefore, the development of such systems involves specific software building processes. These processes are often error-prone because not fully automated, even if some level of automatic code generation or even

model driven engineering support is applied. Furthermore, many critical systems also have assurance requirements, ranging from very strong levels, involving certification (e.g., DO178 and IEC-61508 for safety relevant embedded systems development), to lighter levels based on industry practices.

Over the last two decades, the need for a formally defined safety lifecycle process has emerged. Model-Driven Engineering (MDE) [17] provides a very useful contribution for the design of these systems, since it bridges the gap between design issues and implementation concerns. It helps the designer to specify in a separate way extra-functional requirements at a higher level which are important to guide the implementation process. Of course, an MDE approach is not sufficient but offers an ideal development context. Hence capturing and providing this expertise by the way of specific *patterns* can enhance safety critical systems development. While using an MDE framework, it is possible to help concerns specialists in their task.

In this paper, we investigate the design process of Security and Dependability (S&D) patterns. The main goal of this work is to define a modeling and development framework to support the specifications and the validation of S&D patterns and to assist the developers of trusted applications for resource constrained embedded systems. The solution envisaged here is based on combining metamodeling techniques and formal methods to represent S&D pattern at two levels of abstraction fostering reuse during the process of pattern development and during the process of pattern-based development. The contribution of this work is twofold: (1) An S&D pattern modeling language to get a common representation of S&D patterns for several domains in the context of embedded systems using model driven software engineering, (2) Formal specification and validation of a pattern in order to guarantee its correctness during the pattern integration. Therefore, patterns can be stored in a repository and can be loaded in function of their desired properties. As a result, patterns will be used as bricks to build applications through a model driven engineering approach.

The rest of this paper is organized as follows. An overview of the modeling approach we proposed including a set of definitions is presented in Sect. 2. Then, Sect. 3 presents in detail the pattern modeling language. Section 4 illustrates the pattern modeling process in practice. Section 5 presents the validation process through the example of the authorization pattern. In Sect. 6, we review most related works addressing pattern specification and validation. Finally, Sect. 7 concludes this paper with a short discussion on future works.

2 Conceptual Framework

We promote the separation of general-purpose parts of the pattern from its required mechanisms. This is an important issue to understand the use of patterns to embed solutions targeting extra-functional concerns. This section is dedicated to present the pattern modeling framework. We begin with a set of definitions and concepts that will prove useful in understanding our approach.

2.1 Definitions and Concepts

In [5], a design pattern abstracts the key artifacts of a common design structure that make it useful for creating a reusable object-oriented design. They proposed a set of design patterns for several object-oriented design problems. Several generalizations on this basis to describe software design patterns in general are proposed in literature. Adapting the definition of security patterns given in [18], we propose the following:

Definition 1 (Security and Dependability Pattern). *A security and dependability pattern describes a particular recurring security/and or dependability problem that arises in specific contexts and presents a well-proven generic scheme for its solution.*

Unfortunately, most of S&D patterns are expressed, as informal indications on how to solve some security problems, using identical template to traditional patterns. These patterns do not include sufficient semantic descriptions, including those of security and dependability concepts, for automated processing within a tool-supported development and to extend their use. Furthermore, due to manual pattern implementation use, the problem of incorrect implementation (the most important source of security problems) remains unsolved. For that, model driven software engineering can provide a solid basis for formulating design patterns that can incorporate security and dependability aspects and offering such patterns at several layers of abstraction. We will use metamodeling techniques for representing and reasoning about S&D patterns in model-based development. Note, however, that our proposition is based on the previous definition and on the classical GoF [5] specification, and we deeply refined it in order to fit with the S&D needs.

To foster reuse of patterns in the development of critical systems with S&D requirements, we are building on a metamodel for representing S&D pattern in the form of a subsystem providing appropriate interfaces and targeting S&D properties to enforce the S&D system requirements. Interfaces will be used to exhibit pattern functionality in order to manage its application. In addition, interfaces supports interactions with security primitives and protocols, such as encryption, and specialization for specific underlying software and/or hardware platforms, mainly during the deployment activity. As we shall see, S&D and resource models are used as model libraries to define the S&D and resource properties of the pattern (see part *B* of Fig. 2).

Security and Dependability patterns are not only defined from a platform independent viewpoint (i.e. they are independent from the implementation), they are also expressed in a consistent way with domain specific models. Consequently, they will be much easier to understand and validate by application designers in a specific area. To capture this vision, we introduced the concept of *domain view*. Particularly an S&D pattern at domain independent level exhibits an abstract solution without specific knowledge on how the solution is implemented with regard to the application domain.

Definition 2 (Domain). *A domain is a field or a scope of knowledge or activity that is characterized by the concerns, methods, mechanisms, ... employed in the development of a system. The actual clustering into domains depends on the given group/community implementing the target methodology.*

In our context, a domain represents all the knowledge including protocols, processes, methods, techniques, practices, OS, HW systems, measurement and certification related to the specific domain. With regard to the artifacts used in the system under development, we will identify the first classes of the domain to specialize such artifacts. For instance, the specification of a pattern at domain independent point of view is based on the software design constructs. The specialization of such a pattern for a domain uses a domain protocol to implement the pattern solution (see example of authorization pattern given in Sects. 4.1 and 4.2).

The objective is to reuse the domain independent model S&D patterns for several industrial application domain sectors and also let them be able to customize those domain independent patterns with their domain knowledge and/or requirements to produce their own domain specific artifacts. Thus, the 'how' to support these concepts should be captured in the specification languages.

2.2 Motivational Example: Authorization Pattern

As example of a common and widely used pattern, we choose the *Authorization Pattern* [19]. For instance, in a distributed environment in which users or processes make requests for data or resources, this pattern describes who is authorized to access specific resources in a system, in an environment in which we have resources whose access needs to be controlled. As depicted in the left part of Fig. 1, it indicates how to describe allowable types of accesses (authorizations) by active computational entities (subjects) to passive resources (protection objects).

However, these authorization patterns are slightly different with regard to the application domain. For instance, a system domain has its own mechanisms and means to serve the implementation of this pattern using a set of protocols ranging from RBAC (Role Based Access Control), Firewall, ACLs (Access Control Lists), Capabilities, and so on. For more breadth and depth, the reader is

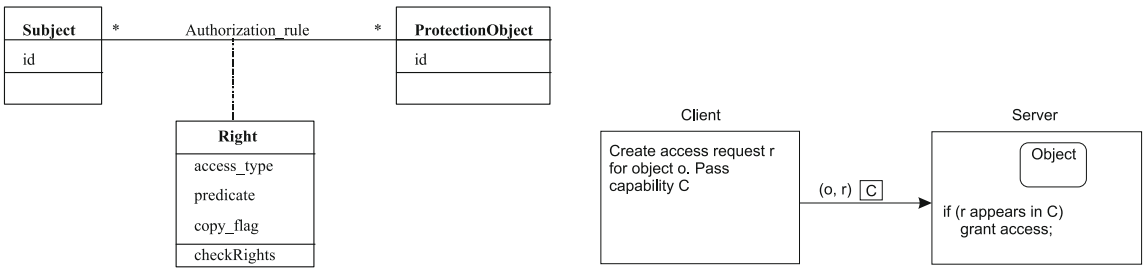

Fig. 1. Authorization pattern/protecting resources using capabilities

referred to [21]. To summarize, they are similar in their goal, but different in the implementation issues. The motivation is to handle the modeling of patterns following different abstraction levels. In the followings, we propose to use *Capabilities* [21] to specialize the implementation of the authorization pattern. This solution is already used at hardware and operating system level to control resources.

More particular, the access rights of subjects with respect to objects are stored in an *access control matrix* M. Each subject is represented by a row and each object is represented by a column. An entry in such a matrix $M[s, o]$ contains precisely the list of operations subject s is allowed to request on object o. A more efficient way to store the matrix is to distribute the matrix row-wise by giving each subject a list of capabilities it has for each object. Without such a capability for a specific object means that the subject has no access rights for that object. Then, requests for resources are intercepted and validated with the information in the capabilities. The interception and the validation are achieved by a special program usually referred to as *reference monitor*. For instance, whenever a subject s requests for the resource r of object o, it sends such a request passing its capability. The reference monitor will check whether it knows the subject s and if that subject is allowed to have the requested operation r, as depicted in the right part of Fig. 1. Otherwise the request fails. It remains the problem of how to protect a capability against modification by its holder. One way is to protect such a capability (or a list of them) with a signature handed out by special certification authorities named *attribute certification authorities*.

2.3 Pattern DSL Building Process and Artifacts

A Domain Specific Language (DSL) typically defines concepts and rules of the domain using a metamodel for the abstract syntax, and a (graphical or textual) concrete syntax that resembles the way the domain tasks usually are depicted. As we shall see, such a process reuses a lot of practices from Model-Driven Engineering (MDE), for instance, metamodeling and transformation techniques. There are several DSML (Domain Specific Modeling Language) environments available. In our context, we use the Eclipse Modeling Framework (EMF) [20] open-source platform. Note, however, that our vision is not limited to the EMF platform.

In Fig. 2, we illustrate the usage of a DSL process based on MDE technology to define the modeling framework to design a pattern on one hand and the use of such a framework on the other hand. As shown in part A of Fig. 2, a DSL process is divided into several kinds of activities: DSL definition, transformation, coherency and relationships rules, designing with DSLs and qualification. The first three activities are achieved by the DSL designer and last two are used by the DSL end-user. In the following we detail the following artifacts and their related processes:

- *Pattern Metamodel:* Sect. 3 (see part A of Fig. 2).
- *Pattern Modeling.* Section 4 (see part A of Fig. 2).
- *Pattern Formalization.* Section 5 (see part C of Fig. 2).

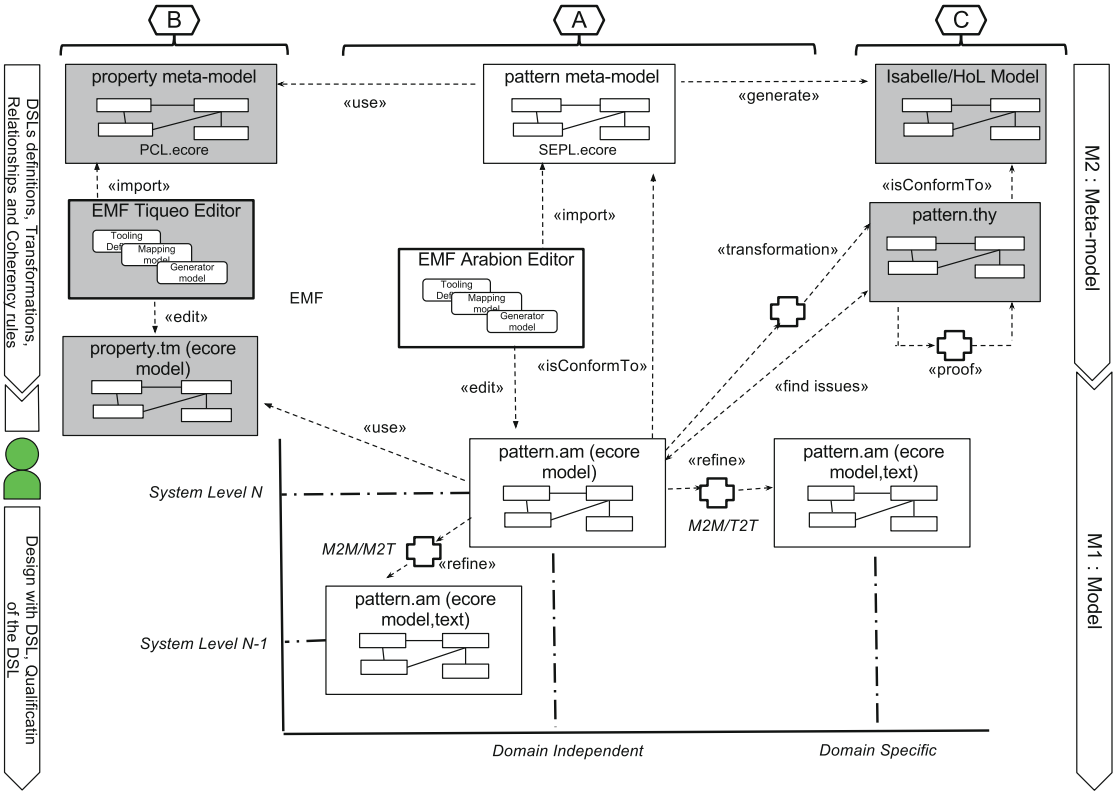

Fig. 2. Overview of the pattern DSL building process and artifacts

3 Pattern Specification Metamodel (SEPM)

The System and Software Engineering Pattern Metamodel (SEPM), as depicted in Fig. 3, is a metamodel which defines a new formalism for describing patterns and which constitutes the base of our pattern modeling language. Such a formalism describes all the concepts (and their relations) required to capture all the facets of patterns. These patterns are specified by means of a domain-independent generic representation and a domain-specific representation.

The principal classes of the metamodel are described with Ecore notations in Fig. 3 and their meanings are more detailed in the following paragraph.

- This block represents a modular part of a system that encapsulates a solution of a recurrent problem. A *DIPattern* defines its behavior in terms of provided and required interfaces. Larger pieces of a system's functionality may be assembled by reusing patterns as parts in an encompassing pattern or assembly of patterns, and wiring together their required and provided interfaces. A *DIPattern* may be manifested by one or more artifacts. This is the key entry artifact to model patterns at domain independent level (DIPM).
- *Interface.* A *DIPattern* interacts with its environment with *Interfaces* which are composed of *Operations.* A *DIPattern* owns provided and required interfaces. A provided interface is implemented by the *DIPattern* and highlights the services exposed to the environment. A required interface corresponds

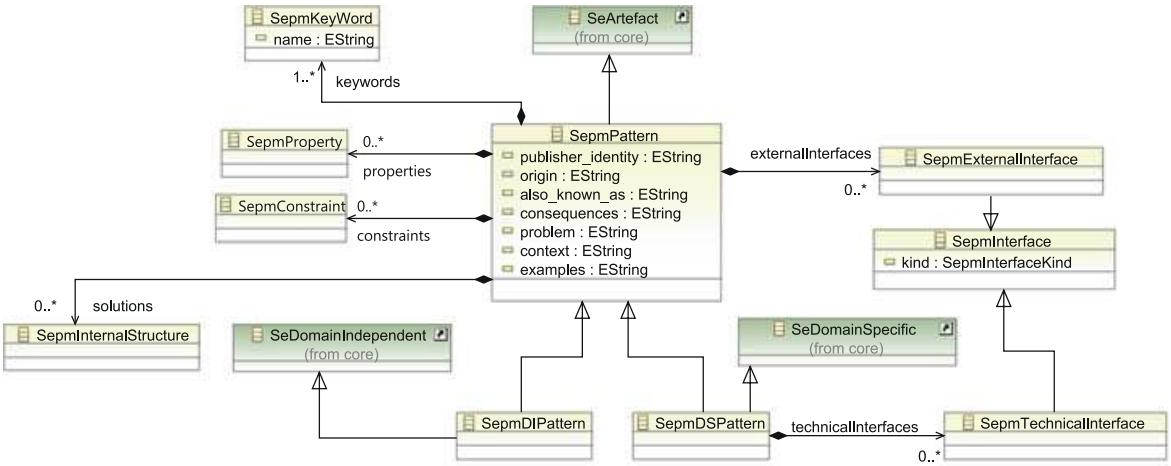

Fig. 3. The SEPM metamodel- overview

to services needed by the pattern to work properly. Finally, we consider two kinds of interface:

- *External interface.* Allows implementing interaction with regard to the integration of a pattern into an application model or to compose patterns.
- *Technical interface.* Allows implementing interaction with the platform. For instance, at a low level, it is possible to define links with software or hardware module for the cryptographic key management. Please note, a *DIPattern* does not have *TechnicalInterfaces*.

– *Property.* Is a particular characteristic of a pattern related to the concern it is dealing with and dedicated to capture its intent in a certain way. Each property of a pattern will be validated at the time of the pattern validating process and the assumptions used, will be compiled as a set of constraints which will have to be satisfied by the domain application.
– *Constraint.* Is a requisite of the pattern. If the constraints are not met, the pattern will not be able to deliver its properties.
– *InternalStructure.* Constitutes the implementation of the solution proposed by the pattern. Thus the *InternalStructure* can be considered as a white box which exposes the details of the *IPatterns*. In order to capture all the key elements of the solution, the *InternalStructure* is composed of two kinds of *Structure*: *static* and *dynamic*. Please, note that the same pattern could have several possible implementations
– *DSPattern.* Is a refinement of a *DIPattern*. It is used to build a pattern at DSPM. Furthermore a *DSPattern* has *TechnicalInterfaces* in order to interact with the platform. This is the key entry artifact to model pattern at domain specific level (DSPM).

3.1 Generic Property Modeling Language (GPRM)

The metamodel of property [23] captures the common concepts of the two main concerns of trusted RCES (Resources-Constrained Embedded Systems)

applications: *Security*, *Dependability* and *Resource* on the one hand and *Constraints* on these properties on the other hand. The libraries of properties and constraints includes units, types, categories and operators. For example, security attributes such as authenticity, confidentiality and availability [1] are categories of S&D properties. The reader is referred to [23] for a full description of the properties metamodel. These models are used as external model libraries to type the properties of patterns. Especially during the design and later on for the selection of the pattern (see next sections), we define the properties and the constraints using these libraries. Note, however, that modeling languages such as MARTE (Modeling and Analysis of Real-Time and Embedded systems) [16] may be used as well for depicting these properties.

4 Pattern Modeling Process

We now present an overview of our pattern modeling process. Along this description, we will give the main keys to understand why our process is based on a general and a constructive approach. It consists of the following phases: (1) the specification of the pattern at domain independent level (DIPM), (2) the refinement of DIPM pattern to specify one of its representations at domain specific level (DSPM). These two levels of the Authorization pattern presented in Sect. 2.2 are illustrated. For the sake of simplicity, many functions and artifacts of this pattern have been omitted. We only detail the properties and interfaces that we need to describe the validation process. Note that the document representing the formalization and the proof are detailed in Sect. 5.

4.1 Domain Independent Pattern Model (DIPM)

This level is intended to generically represent patterns independently from the application domain. This is an instance of the SEPM. As we shall see, we introduce new concepts through instantiation of existing concepts of the SEPM metamodel in order to cover most existing patterns in safety critical applications. In our example, the DIPM of the authorization pattern consists of:

- *Properties.* At this level, we identify one property: *confidentiality.*
- *External Interfaces.* The authorization pattern exposes its functionalities through function calls:
  - $req(S, AT, PR)$: the subject S sends request about access type AT concerning the protected resource or data PR.

4.2 Domain Specific Pattern Model (DSPM)

The objective of the specific design level is to specify the patterns for a specific application domain. This level offers artifacts at down level of abstraction with more precise *information* and *constraints* about the target domain. This modeling level is a refinement of the DIPM, where the specific characteristics and

dependencies of the application domain are considered. Different DSPM would refine a same DIPM for all needed domain. For instance, when using *Capabilities* as a mechanism related to the application domain to refine the authorization pattern at DSPM, we identify the following artifacts:

- *Properties.* In addition to the refinement of the property of confidentiality identified in the DIPM, at this level, one can consider: *deny unauthorized access*, *permit authorized access*, and *efficiency* properties.
- *External Interfaces.* This is a refinement of the DIPM external interface: $req(S, AT, PR, C)$: the subject S sends request about access type AT concerning the protected resource PR passing its capability C.
- *Technical Interfaces.* Let the subset of functions related to the use of capabilities to refine the authorization pattern:
  - $sign(C)$: the certification authority signs the capability C,
  - $verifyCert()$: the attribute capability certificate is verified,
  - $extractCap()$: the capability is extracted from the certificate,
  - $checkRight(S, AT, PR, C)$: the reference monitor verifies, using the capability, whether PR appears in the C.

5 Pattern Validation Process

We propose to use theorem proving techniques to formalize patterns and to prove their properties. This part completes the framework as depicted in Fig. 2 (see Part C). In our work, we used the interactive Isabelle/HOL proof assistant [15] to do so. First we model the pattern as a data type. Then we define and prove a set of properties and constraints that it meets. These two steps are applied successively to DIPM and DSPM levels. Finally, we deal with the correspondence between the DIPM and DSPM formal models.

5.1 Pattern Formalization

Before coding the pattern and its properties in Isabelle/HOL, one has to define what are an agent and an action. Both are records i.e. tuples, with a name attached to each component. We note *i_action* the record for an action defined at the DIPM level, and we will talk about *s_action* when dealing with the specific level. The right part of the action data type with "..." refers to specific actions defined at the DSPM model.

Definitions mainly concern the refinement process of pattern from the DIPM model to the DSPM model. We formalize this refinement by adding to the definitions of DIPM a subset of functions related to the mechanisms employed in the application domain. Compared to independent actions embodied by the *i_action* records at the DIPM level of the pattern, records of type *s_action* devoted to DSPM may have additional fields. This is easily done by the *s_action* definition of which extends the first specification by the keyword + (see left part of Table 1). We so consider that the DSPM formal model contains the same set

Table 1. Data types for the authorization pattern

DIPM	DSPM
`datatype agentType = S`	
`datatype actionType = req | ...`	
`datatype accessType = at`	
`datatype dataType = PR`	
	`datatype capabilityType = C`
`record i_action =`	`datatype actionType = req | ... | checkRight`
`agent : : "agentType"`	
`access : : "accessType"`	`record s_action = i_action +`
`data : : "dataType"`	
	`capability : : "capabilityType"`
`record agent =`	
`name : : "agentType"`	
`i_actions : : "i_action set"`	
`s_actions : : "s_action set"`	

of agents as the DIPM one. Its actions correspond to the external and internal domain specific interface function calls.

For our case, we introduce how to specify the confidentiality property. Let G be a set of agents, D be the critical data and A be the actual set of actions that use D. Then, the confidentiality property on D, denoted by $conf(A, D, G)$, means that only agents in G are allowed to know the value of D. All others, when looking at actions in A shall not be able to derive its value.

The definitions and the proofs used in this section are extracted from the code of our experiment applied to the authorization pattern and its related mechanism from the application domain: capabilities. Table 1 defines these artifacts in Isabelle.

Domain Independent Pattern Model (DIPM). As previously defined in Sect. 4.1, the authorization pattern exposes its functionalities through the function calls $req(S, AT, PR)$. In this formula, the subject S sends requests about access type AT concerning the protected data PR. Applying the previous definition $conf$, we get $A = req(S, at, PR)$ where $at \in AT$, $D = PR$ and $G = S$.

Function calls $req(S, at, PR)$ are independent requests of the authorization pattern formalized in Isabelle as shown in the left part of Table 2 owing to the *create_i_Action* definition and the *req* constant. In the same way, an agent is introduced by the *createAgent* definition and the subject constant nominates the active computational entity of the pattern.

Domain Specific Pattern Model (DSPM). The DSPM model of the authorization pattern introduces a capability C as mechanism to ensure the communication between the subject and the passive resources. Consequently, the DSPM $req(S, AT, PR, C)$ external interface is a refinement of the DIPM one where

Table 2. Definitions of the authorization pattern

DIPM

```
definition create_i_Action : :
"agentType ==> accessType ==> dataType
==> i_action"
where
"create_i_Action ag a d == (agent=ag,
access=a, data=d)"

definition req : : "i_action" where
"req == create_i_Action S at PR"

definition createAgent : :
"agentType ==> (i_action set)
==> (s_action set) ==> agent"
where
"createAgent ag i_as s_as ==
(name=ag, i_actions=i_as, s_actions=s_as)"

definition subject : : "agent" where
"subject == createAgent S req
checkRight, ..."
```

DSPM

```
definition create_s_Action : :
"agentType ==> accessType
==> dataType ==> capabilityType
==> s_action" where
"create_s_Action ag a d c ==
(agent=ag, access=a, data=d,
capability=c)"

definition checkRight : : "s_action"
where
"checkRight == create_s_Action S
at PR C"
```

the subject S sends requests about access type AT concerning the protected resource PR passing its capability C. We formalize this refinement by the code in the right part of Table 1 which adds to the definitions of the DIPM part the subset of functions *sign*, $verifyCert$, $extractCap$ and $checkRight$ related to capabilities. This code also defines the counterpart *s_action* of an independent action *i_action*. For sake of clarity, we only give the code of the $checkRight$ constant defined as an *s_action*.

Following the definition of Sect. 4.1 and applying the previous definition $conf$, we get the confidentiality property $conf(checkRight(S, at, PR, C), PR, S)$.

5.2 Pattern Validation

The goal now is to identify the set of assumptions required to prove the properties. Then, the proof consists to find a scheduling of valid steps using Isabelle's tactics such as applying (*command apply*) a simplification considering that each step corresponds to a sub-goal to resolve. Our proof only uses simplification rules (*command simp*) which consist in rewriting specified equations from left to right in the current goal. Correctness of the proof is guaranteed by construction from the first goal to resolve (a lemma or a theorem) until the message *"no sub-goals"* is produced by the framework which confirms that the proof is finished (*command done*).

Domain Independent Pattern Model (DIPM). For our case, we have to find the assumptions for the $conf(req(S, at, PR), PR, G)$ property to hold, supposing $S \in G$ and $at \in AT$. Informally, to achieve such a property, we need to demonstrate that G is the only set of agents that can access to PR using an action $req(S, at, PR)$ and that such an action doesn't provide any information to an agent outside of G in order to derive the value of the protected data. The pattern describes in fact who is authorized to access specific resources in a system and whose accesses need to be controlled.

By definition, as presented in Sect. 2.2, we have a vector $M[s1..sn][PR]$ to denote the access rights. An entry in such a matrix $M[si, PR]$ contains precisely the list of operations of AT subject si that are allowed to request on the object PR. At this level, it's obvious that the only agents allowed to use the protected data are those listed in the defined matrix. For the second concern, we have to assume that the matrix is protected itself and that agents outside of G looking at actions in $req(S, at, PR)$ shall not be able to derive the value of PR.

Table 3. Proving the conf property

DIPM	DSPM
`definition isAllowed : :`	
`"i_action ==> dataType ==> agent`	
`==> bool" where`	
`"isAllowed a d ag == a in (i_actions ag)`	
`and data a = d"`	
`definition isAllowedSubject : : "bool"`	
`where`	
`"isAllowedSubject == isAllowed req`	
`PR subject"`	
	`definition conf_s_action : :`
`definition conf_i_action : :`	`"s_action ==> dataType ==> agent`
`"i_action ==> dataType ==> agent`	`==> bool"`
`==> bool" where`	`where`
`"conf_i_action a d ag ==`	`"conf_s_action a d ag == a`
`isAllowed a d ag and isProtected a d ag"`	`in (s_actions ag)`
	`and data a = d"`
`lemma confReq : "conf_i_action req`	
`PR subject"`	`lemma confCheckRight :`
`apply (simp only : conf_i_Action_def)`	`"conf_s_action checkRight`
`apply (simp only : isAllowed_def`	`PR subject"`
`isProtected_def)`	
`apply (simp only : req_def)`	
`apply (simp only : create_i_Action_def)`	
`apply (simp only : subject_def)`	
`apply (simp only : createAgent_def)`	
`apply (simp add : req_def)`	
`apply (simp only : create_i_Action_def)`	
`done`	

In our Isabelle experiment, we have weakened these constraints assuming two predicates *isAllowed* for authorized accesses to a specific resource and *isProtected* related to its encapsulated access. We simulate the matrix by a set of actions. We also ensure that the term $req(S, at, PR)$ verifies these preconditions. With these assumptions, the relation $conf(req(S, at, PR), PR, S)$ holds as proved by the *lemma confReq* (confidentiality of the req request) of the left part of Table 3.

In the next section, using a specific mechanism, namely *Capabilities*, to implement such a pattern, we will refine these assumptions.

Domain Specific Pattern Model (DSPM). As an example, the confidentiality property $conf(checkRight(S, at, PR, C), PR, S)$ is then established by the *confCheckRight lemma* (confidentiality of the checkRight request) given in the right part of Table 3, with similar simplification tactics when proving the $confReq$ lemma of the left part of Table 3.

For our experiment, no assumptions have been introduced for this proof, but we can easily extend our formalization by considering for instance a precedence order between the calls *sign*, $verifyCert$, $extractCap$ and *checkRight* related to capabilities.

5.3 Correspondence between DIPM and DSPM

Correspondence between the DIPM and DSPM formal models is assumed when proving that the property introduced at the DIPM model is transferred to the DSPM model. More precisely, this means that the DIPM model is an abstraction of the DSPM model. In particular, we must show that using a specific mechanism for verifying the property together with function calls of the specific domain is a specific case of proving the upper-level property.

We so have to map actions of the DSPM model onto the actions of the DIPM model by an appropriate homomorphism h and then prove that this homomorphism preserves the property. As a property is proved both for the DIPM and DSPM models, h validates the refinement of the proof. In practice, h is required to preserve each operation or a pseudo-operation which summarizes the behavior of a set of operations.

In our case, we must show that using *capabilities* for verifying the property $conf(checkRight(S, at, PR, C), PR, S)$ together with function calls of the specific domain is a specific case of proving the upper-level property $conf(req(S, at, PR), PR, S)$. Fig. 4 specifies h and the resulting $confReqH$ *theorem* (confidentiality of the req request using h). In this code, for simplicity of illustration, we only map *checkRight* to req, and we introduce a null action for all other mappings.

Hence h preserves confidentiality into the DIPM request access to the protected data. The $conf(req(S, at, PR), PR, S)$ relation transferred to the DSPM model is identical to $conf(checkRight(S, at, PR, C), PR, S)$. We can formally verify this assumption according to the Isabelle's framework which displays the same sub-goals to resolve as soon as the DSPM current goal of Fig. 4 has been split by a case distinction on a Boolean condition (*command simp* with *if_def*).

```
 definition h :: "s_action ==> i_action" where
"h a == if a = checkRight then req else null"

theorem confReqH :
"conf_i_action (h checkRight) PR subject"
apply (simp only : conf_i_action_def)
apply (simp only : h_def)
apply (simp only : if_def)
apply (simp)
apply (simp only : req_def)
apply (simp only : create_i_Action_def)
apply (simp only : subject_def)
apply (simp only : createAgent_def)
apply (simp add : req_def)
apply (simp only : isAllowed_def isProtected_def)
apply (simp only : create_i_Action_def)
apply (simp)
done
```

Fig. 4. Proving the correspondence between DIPM and DPSM models

6 Related Works

Design patterns are a solution model to generic design problems, applicable in specific contexts. Supporting research tackles the presented challenges includes domain patterns, pattern languages and recently formalisms and modeling languages to foster their application in practice. To give an idea of the improvement achievable by using specific languages for the specification of patterns, we look at pattern formalization and modeling problems targeting the integration of the pattern specification and validation steps into a broader MDE process.

Several tentatives exist in the literature to deal with patterns for specific concern [7,22]. They allow to solve very general problems that appear frequently as sub-tasks in the design of systems with security and dependability requirements. These elementary tasks include secure communication, fault tolerance, etc. The pattern specification consists of a service-based architectural design and deployment restrictions in form of UML deployment diagrams for the different architectural services.

To give a overview of the improvement achievable by using specific languages, we look at the pattern specification and formalization problems. *UMLAUT* [8] is an approach that aims to formally model design patterns by proposing extensions to the UML metamodel 1.3. They used OCL language to describe constraints (structural and behavioral) in the form of meta-collaboration diagrams. In the same way, *RBML(Role-Based Metamodeling Language)* [12] is able to capture various design perspectives of patterns such as static structure, interactions, and state-based behavior.

While many patterns for specific concern have been designed, still few works propose general techniques for patterns. For the first kind of approaches [5], design patterns are usually represented by diagrams with notations such as UML object, annotated with textual descriptions and examples of code. There are some well-proven approaches [3] based on Gamma et al. However, this kind of

technique does not allow to reach the high degree of pattern structure flexibility which is required to reach our target.

Formal specification has also been introduced in [14] in order to give rigorous reasoning of behavioral features of a design pattern in terms of high-level abstractions of communication. In this paper, the author considers an object-oriented formalism for reactive system (DisCo) [11] based on TLA (Temporal Logic of Actions) to express high-level abstractions of cooperation between objects involved in a design pattern. However, patterns are directly formalized at the pattern level including its classes, its relations and its actions, without defining a metamodel.

[6] presents a formal and visual language for specifying design patterns called LePUS. It defines a pattern in an accurate and complete form of formula in Z, with a graphical representation. With regard to the integration of patterns in software systems, the DPML (Design Pattern Modeling Language) [13] allows the incorporation of patterns in UML class models. However, this kind of techniques does not allow to achieve the high degree of pattern structure flexibility which is required to reach our target, and offered by the proposed pattern modeling language. The framework promoted by LePUS is interesting but the degree of expressiveness proposed to design a pattern is too restrictive.

Regarding the analysis aspects, [10] used the concept of security problem frames as analysis patterns for security problems and associated solution approaches. They are also grouped in a pattern system with a list of their dependencies. The analysis activities using these patterns are described with a highlight of how the solution may be set, with a focus on the privacy requirement anonymity. For software architecture, [9] presented an evaluation of security patterns in the context of secure software architectures. The evaluation is based on the existing methods for secure software development, such as guidelines as well as on threat categories.

Another important issue is the identification of security patterns. [2] proposed a new specification template inspired on secure system development needs. The template is augmented with UML notations for the solution and with formal artifacts for the requirement properties. Recently [4] presented an overview and new directions on how security patterns are used in the whole aspects of software systems from domain analysis to the infrastructures.

To summarize, in software engineering, design patterns are considered effective tools for the reuse of specific knowledge. However, a gap between the development of systems using patterns and the pattern information still exists. This becomes even more visible when dealing with specific concerns namely security and dependability for several application sectors.

– From the pattern methodological point of view: The techniques presented above do not allow to reach the high degree of pattern structure flexibility. The framework promoted by LePUS is interesting but the degree of expressiveness proposed to design a pattern is too restrictive. The main critic of these modeling languages is that no variability support of the pattern specification, because a pattern by nature covers not only one solution, but describes

a set of solutions for a recurring design problem. Furthermore there are no elements in existing modeling languages to model appropriate architectural concepts and properties provided by patterns and even more security and dependability aspects. The problem is obvious in UML and ADLs. The main shortcoming of the UML collaboration approach stems from the non-support of variability. However, we do believe that the advantages of using UML for engineering software outweigh these disadvantages.

- From the pattern-based software engineering methodological point of view: Few works are devoted to this concern. They are in line for the promotion of the use of patterns in each system/software development stage. However, existing approaches using patterns often target one stage of development (architecture, design or implementation) due to the lack of formalisms ensuring both (1) the specification of these artifacts at different levels of abstraction, (2) the specification of relationships that govern their interactions and complementarity and (3) the specification of the relationship between patterns and other artifacts manipulated during the development lifecycle and those related to the assessment of critical systems.
- From the pattern validation process point of view: There are mainly two approaches when dealing with formal reasoning: model-checking and interactive theorem proving. Model-checking is attractive because it offers a high degree of automation and is therefore accessible to uninitiated users. However, model checkers have to provide an intuitive operational understanding of a model under properties they have to verify. In contrast theorem proving focuses on abstract properties on a model and not on its behavior. This is the case for our pattern validation process with Isabelle/HOL: proof obligations introduce formulas that are so many assumptions to validate, and DSPM actions mapped on to DIPM actions must normally encompass the same subset of proof obligations.
- From the tool support point of view: Existing tools support the specification of patterns and their application in the design of software systems, notably Netbeans, StarUML, Sparx systems. In these tools, patterns are provided as UML libraries and usually embedded in the tool without extension support. The integration of pattern in these tools result merely in copying a solution into a model.

7 Conclusion and Future Work

We present an approach for the design and verification of patterns to provide practical solutions to meet security and dependability (S&D) requirements. As it follows the MDE paradigm for system's design, using patterns on different levels of abstraction, it allows for integration into the system's design process, hence supports this process. To this end, the proposed representation takes into account the simplification and the enhancement of such activities, namely : selection/search based on the properties, and integration based on interfaces. Yet the system development process is not the topic we focus on in our paper.

Indeed, a classical form of pattern is not sufficient to tame the complexity of safety critical systems – complexity occurs because of both the concerns and the domain management. To reach this objective and to foster reuse, we introduced the specification at domain independent and domain specific levels. The former exhibits an abstract solution without specific knowledge on how the solution is implemented with regard to the application domain. Following the MDE process, the domain independent model of patterns is then refined towards a domain specific level, taking into account domain artifacts, concrete elements such as mechanisms to use, devices that are available, etc.

We also provide an accompanying formalization and validation framework to help precise specification of patterns based on the interactive Isabelle/HOL proof assistant. The resulting validation artefacts may mainly (1) complete the definitions, and (2) provide semantics for the interfaces and the properties in the context of S&D. Like this, validation artefacts may be added to the pattern for traceability concerns. In the same way, the domain refinement is applied during the formal validation process for the specification and validation of patterns.

Furthermore, we walk through a prototype of EMF tree-based editors supporting the approach. Currently the tool suite named *Semcomdt*[1] is provided as Eclipse plugins. The approach presented here has been evaluated on two case studies from TERESA's project[2] resulting in the development of a repository of S&D patterns with more than 30 S&D patterns. By this illustration, we can validate the feasibility and effectiveness of the proposed specification and design framework.

The next step of this work consists in implementing other patterns including those for security, safety, reconfiguration and dependability to build a repository of multi-concerns patterns. Another objective for the near future is to provide guidelines and tool-chain supporting the whole pattern life cycle (i.e., create, store patterns, retrieve) and the integration of pattern in an application. All patterns are stored in a repository. Thanks to this, it is possible to find a pattern regarding to concern criteria. At last, guidelines will be provided during the pattern development and the application development (i.e., help to choose the appropriate pattern and its usage).

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**, 11–33 (2004)
2. Cheng, B., Cheng, B.H.C., Konrad, S., Campbell, L.A., Wassermann, R.: Using security patterns to model and analyze security. In: IEEE Workshop on Requirements for High Assurance Systems, pp. 13–22 (2003)
3. Douglass, B.P.: Real-time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley, Reading (1998)
4. Fernandez, E.B., Yoshioka, N., Washizaki, H., Jürjens, J., VanHilst, M., Pernul, G.: Using security patterns to develop secure systems. In: Mouratidis, H. (ed.) Software Engineering for Secure Systems: Industrial and Research Perspectives, pp. 16–31. IGI Global (2010)

[1] http://www.semcomdt.org
[2] http://www.teresa-project.org/

5. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
6. Gasparis, E., Nicholson, J., Eden, A.H.: LePUS3: an object-oriented design description language. In: Stapleton, G., Howse, J., Lee, J. (eds.) Diagrams 2008. LNCS (LNAI), vol. 5223, pp. 364–367. Springer, Heidelberg (2008)
7. Di Giacomo, V., et al.: Using security and dependability patterns for reaction processes. In: Proceedings of the 19th International Conference on Database and Expert Systems Application, pp. 315–319. IEEE Computer Society (2008)
8. Le Guennec, A., Sunyé, G., Jézéquel, J.-M.: Precise modeling of design patterns. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 482–496. Springer, Heidelberg (2000)
9. Halkidis, S.T., Chatzigeorgiou, A., Stephanides, G.: A qualitative analysis of software security patterns. Comput. Secur. **25**(5), 379–392 (2006)
10. Hatebur, D., Heisel, M., Schmidt, H.: A security engineering process based on patterns. In: Proceedings of the 18th International Conference on Database and Expert Systems Applications, DEXA '07, pp. 734–738. IEEE Computer Society, Washington (2007)
11. Jarvinen, H.M., Kurki-Suonio, R.: Disco specification language: marriage of actions and objects. In: 11th International Conference on Distributed Computing Systems, pp. 142–151. IEEE Press (1991)
12. Kim, D.K., France, R., Ghosh, S., Song, E.: A UML-based metamodeling language to specify design patterns. In: Patterns, Proceedings Workshop Software Model Engineering (WiSME) with Unified Modeling Language Conference 2004, pp. 1–9 (2004)
13. Mapelsden, D., Hosking, J., Grundy, J.: Design pattern modelling and instantiation using dpml. In: CRPIT '02: Proceedings of the Fourteenth International Conference on Tools Pacific, pp. 3–11. Australian Computer Society Inc. (2002)
14. Mikkonen, T.E.: Formalizing design patterns. In: Proceeding ICSE '98 Proceedings of the 20th International Conference on Software Engineering. IEEE Press (1998)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
16. OMG. OMG. A UML profile for MARTE: modeling and analysis of real-time embedded systems, beta 2. (June 2008). http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf
17. Schmidt, D.: Model-driven engineering. IEEE Comput. **39**(2), 41–47 (2006)
18. Schumacher, M.: Security Engineering with Patterns - Origins, Theoretical Models, and New Applications. LNCS, vol. 2754. Springer, Heidelberg (2003)
19. Schumacher, M., Fernandez, E., Hybertson, D., Buschmann, F.: Security Patterns: Integrating Security and Systems Engineering. Wiley, Chicester (2005)
20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0., 2nd edn. Addison-Wesley Professional, Reading (2009)
21. Tanenbaum, A.S., Steen, M.: Distributed systems, principles and paradigms, 2/E. Prentice-Hall Inc., Upper Saddle River (2007)
22. Yoshioka, N., Washizaki, H., Maruyama, K.: A survey of security patterns. Prog. Inform. **(5)**, 35–47 (2008)
23. Ziani, A., Hamid, B., Trujillo, S.: Towards a unified meta-model for resources-constrained embedded systems. In: 37th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 485–492. IEEE (2011)